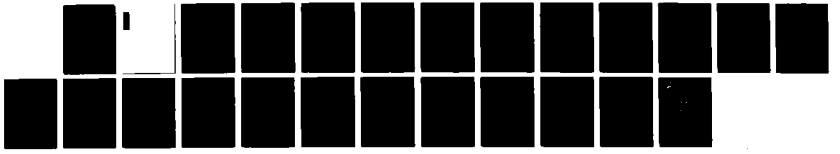
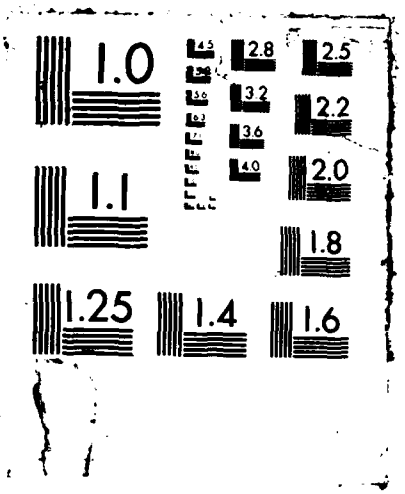
AD-A189 427 OBJECT-ORIENTED DESIGN IN NUMERICAL LINEAR ALGEBRA(U) 1/1
WASHINGTON UNIV SEATTLE DEPT OF STATISTICS
J A MCDONALD SEP 87 TR-109 N00014-86-K-0069

UNCLASSIFIED F/G 12/1 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5

```
(defmethod mul ((tr Inverse-of-Upper-Triangular)
                (v Vector))

  (loop with r = (make-array (range tr))
        with u = (matrix (sire tr))
        with n = (domain tr)

        ;; back substitution loop:
```

Object-oriented design in numerical linear algebra. *

JOHN ALAN MCDONALD
Dept. of Statistics, University of Washington

September 6, 1987

Abstract

Straightforward application of object-oriented design to standard algorithms in numerical linear algebra improves clarity and expressiveness, without sacrificing speed or accuracy.

*This research was supported by the Office of Naval Research under Young Investigator award N00014-86-K-0069, the Dept. of Energy under contract FG0685-ER2500.

1 Introduction

The underlying premise of this paper is that quantitative, scientific computing needs and deserves good programming languages and programming environments—as much as artificial intelligence.

I hope to show how straightforward application of object-oriented design to standard algorithms in numerical analysis yields immense improvement in clarity, without sacrificing speed or accuracy. My example is an elementary problem from numerical linear algebra—solving systems of linear equations via gaussian elimination. I will use a linear algebra system called *Cactus*, which is implemented in Common Loops[2,3], an object-oriented extension of Common Lisp[14].

Object-oriented programming is sometimes said to only be useful for graphics and user interface and, perhaps also, knowledge representation and database management. Even Lisp machine manufacturers seem to think that Fortran is somehow superior for traditional, quantitative scientific computing. When numerical software is written in Lisp (eg. [12]), the author usually adopts a Fortran or Algol style and neglects the potential for designing more appropriate abstractions[1].

Cactus is based on a collection of *linear transformation* classes and appropriate generic operations. This level of abstraction greatly simplifies many algorithms in numerical linear algebra. Traditional linear algebra systems (Linpack[5], APL) operate at the level of arrays and confound the details of where data is kept with how it is meant to be used.

I am developing Cactus primarily as a building block of a Lisp-based scientific computing environment. In addition, I hope that it will lead to more serious consideration for numerical computing in Lisp (and Smalltalk[7], etc.) and help to put to rest the notion of *coupled systems*, where symbolic computing is done in Lisp and serious numerical computing is done in another language (or even on another machine). The idea of coupled systems arises because, although existing Lisp environments have much to offer scientific computing, they have two serious deficiencies when compared to more traditional environments (eg. Unix plus C and Fortran):

- No analogs of numerical subroutine libraries like Linpack, Eispack, etc. [5,13,6].
- Poor performance in arithmetic, especially with double precision or complex numbers.

Building collections of basic numerical software should be done by experts, which requires convincing numerical analysts that there are sufficient advantages that it is worth re-implementing (and re-designing!) the existing subroutine libraries. However, I will present the case to numerical analysts in another paper.

This paper is instead aimed at the designers and implementors of object-oriented programming systems. One purpose is to convince them to provide efficient arithmetic. But perhaps more interesting is the fact that linear algebra is a fairly novel domain for object-oriented programming. As a result, Cactus provides another perspective on what's important in an object-oriented programming system. In particular, numerical linear algebra requires some of the relatively unusual features of Common Loops—including methods that dispatch on several arguments, changing or modifying the class of an existing instance, and specialized method combination rules—which are missing or less well supported in older object-oriented languages like Smalltalk or Flavors[10,11].

This paper assumes the reader is familiar with Lisp and object-oriented programming, specifically as embodied in Common Loops. I will review, in passing, some elementary numerical linear algebra.

2 A simple problem

Suppose

$$A : \mathcal{R}^n \mapsto \mathcal{R}^m$$

is a linear transformation, $b \in \mathcal{R}^m$ and we want to solve for $x \in \mathcal{R}^n$ such that:

$$Ax = b.$$

A mathematician would probably view this as a vector equation. If—to make things easy—we assume that A is square and invertible, then the obvious, naive solution is to invert A and apply it to b:

$$x \leftarrow A^{-1}b.$$

Unfortunately, this isn't such a good idea in floating point arithmetic[5,8,15]. The problem is that we can only compute an approximation $\tilde{A}^{-1}$. For some A's,

$$\tilde{x} \leftarrow \tilde{A}^{-1}b$$

```
(setf a #2a((2.0 3.0e6 9.0e6 1.0e6)
            (8.0 3.0e6 2.0e6 1.0e6)
            (2.0 9.0e6 1.0e6 1.0e6)
            (3.0 5.0e6 7.0e6 9.0e6)))

(setf x #(0.0 1.0 100.0 10000.0))

(setf b (matrix-multiply a x))

(setf a-1 (matrix-inverse a))

(setf x-tilde (matrix-multiply a-1 b))

x-tilde → #(-64.0 0.9993286 100.00006 10000.002)
```

Figure 1: An example of unstable inversion.

maybe be far from the true solution x. See figure 1 for an example.

To get reliable solutions, every numerical linear algebra text recommends:

- Avoid computing inverses.
- Think in terms of solving a system of linear *scalar* equations, rather than a single vector equation. (Linpack is described not as a linear algebra package, but as "a collection of Fortran subroutines which analyze and solve various systems of simultaneous linear equations."[5])
- Notice that we can directly solve systems of linear equations with special structure:

  If U is upper triangular ($U_{i,j}$ is zero if $i > j$). then

  $$U_{n,n}x_n = b_n,$$

  $$U_{n-1,n-1}x_{n-1} + U_{n-1,n}x_n = b_{n-1},$$

  and so on, which means that we can first solve for x_n and substitute it in the second equation to get x_{n-1} and so on. This is called *back substitution*.

  If L is lower triangular ($L_{i,j}$ is zero if $j > i$), then we can solve $Lx = b$ with a similar *forward elimination*.

- If we can decompose $A \rightarrow LU$, then we can solve $Ax = b$ in two steps, first solving $Ly = b$ and then $Ux = y$.
- Then the problem is to find an accurate and efficient way to carry out the LU decomposition.

  The usual recommendation is to use *gaussian elimination with partial pivoting*. The "partial pivoting" is introduced to improve numerical stability. Instead of the simple LU decomposition, we get something a bit more complicated:

  $$A \rightarrow (P_0G_0P_1G_1 \ldots P_nG_nU).$$

  where the P_k's are pivot transformations (permutations) and the G_k's are elementary lower triangular (gauss) transformations. So, conceptually, the procedure is to solve a sequence of linear systems. The system involving U is solved by back substitution; the ones involving P_k and G_k can be solved more directly.

```
(multiple-value-setq (p lu) (lu-decomposition a))

p → #(1 2 0 3)
lu → #2A((0.25 0.27272728 8363636.5 545454.56)
         (8.0 3000000.0 2000000.0 1000000.0)
         (0.25 8250000.0 500000.0 750000.0)
         (0.375 0.46969697 0.7192029 7880435.0))

(lu-solve p lu b) → #(0.0 0.9999205 100.000015 10000.0)

;; The true answer:
x                 → #(0.0 1.0000000 100.000000 10000.0)
```

Figure 2: Using a Fortran-style LU decomposition.

In addition to improved numerical stability, this approach is 2-3 times as fast as explicitly computing the inverse matrix and then using matrix multiply to apply it to b (because computing the inverse matrix is equivalent to solving several systems of equations).

Figure 2 shows what happens when we use an LU decomposition on the data from figure 1. (Although the example is expressed in Lisp, it has essentially the same structure as corresponding Fortran routines in Linpack.)

Although we are happy to get the right answer, we may be troubled by the mysterious arrays p and lu:

- p is the permutation vector that corresponds to the composition of the P_k's. Factoring out the P_k's in this way relies on subtleties in the way they are constructed and related to the G_k's.
- lu packs together the G_k's and U in one array. The upper triangle of lu is U; The kth sub-diagonal column of lu holds the non-zero elements of a vector α_k (whose first k elements are implicitly zero) that can be

used to construct the Gauss matrices G_k:

$$G_k = I + \alpha_k e_k^\dagger$$

(e_k is the kth canonical basis vector of $\mathcal{R}^n$. lu-solve uses the α_k's directly and never explicitly forms the corresponding G_k.

3 What's wrong with this?

There are two problems with even the best Fortran subroutine libraries (like Linpack):

- It is difficult to modify a routine like lu-decomposition to produce something slightly different.
- It is difficult to use the results of lu-decomposition in a way that's different from what the designer intended.

Both cases require understanding how the contents of the two arrays, p and lu are meant to be interpreted. This information is buried in the code for *lu-decomposition* and lu-solve. Understanding the code for an LU decomposition requires following a trail that starts with a abstract, often geometric, overview in a numerical analysis text, through a fairly clean "implementation" in Algol-like pseudo-code, to finally the dirty realities of Fortran IV control structures and parameter passing tricks. Information about the meaning of what going on is stripped out at each stage. The final Fortran program has very little resemblance to the abstract description we started with.

Of course, Linpack, Eispack, etc. were never intended to be anything else other than kits of black boxes. One sometimes hears the argument that numerical linear algebra is sufficiently well understood that there is never reason to modify the existing libraries. Even if this were true for linear algebra, it's definitely not the case for important applications of linear algebra, like optimization or statistical data analysis, and that require adjusting, tuning, modifying, or re-building algorithms for each specific problem.

4 Linear transformations

The primary reason why traditional numerical packages are difficult to modify is that they are not written at the right level abstraction (which unavoidable when working in Fortran). In the case of linear algebra this means the failure to distinguish the abstract notion of a *linear transformation* [9] from its concrete representation as an array or some part of an array.

By definition, A is a *linear transformation* taking $\mathcal{R}^n \mapsto \mathcal{R}^m$, if, for all $x, y \in \mathcal{R}^n$ and $a, b \in R$,

$$A(ax + by) = aAx + bAy.$$

A machine representation of a linear transformation must provide an encoding of its state and methods for essential operations:

- Transformation of a vector: $Ax = y$.
- Scalar multiplication: $(aA)x = A(ax)$.
- Sum: $(A + B)x = Ax + Bx$.
- Composition: $(AB)x = A(Bx)$.

Although "matrix algebra" is often used interchangeably with "linear algebra," in algorithms like the LU decomposition, matrices serve only as a storage mechanism, and an inconsistantly used storage mechanism at that. The canonical representation of a linear transformation is the matrix of its elements (in the canonical basis for $\mathcal{R}^n$) and the composition method, for example, is matrix multiply. However, some linear transformations are better served by an alternate representation. The key entities in an LU decomposition are all linear transformations, but not all are matrices:

- A is represented by the matrix a.
- U is represented by the upper triangle of lu.
- G_k is represented by the sub-columns of lu.
- $\prod P_k$ is represented by p.

The natural way to solve $Ax = b$, is using an inverse: $x \leftarrow A^{-1}b$ (assuming, again to make things easy, that A is both square and invertible). It is bad practice to do this if both A and A^{-1} must be matrices, but that's no longer true if we can represent A^{-1} by an instance of an appropriate Inverse-Transformation class. The state of A^{-1} would be represented by something equivalent to the lu and p produced by lu-decomposition and A^{-1}'s method for transforming a vector would be equivalent to lu-solve.

5 Implementation in Common Loops

Cactus deals only with transformations from $\mathcal{R}^n \mapsto \mathcal{R}^m$. Linear transformations are represented in Cactus by instances of a Common Loops linear transformation class.

Two examples of simple linear transformation classes are given in figure 3. All linear transformation classes inherit from the abstract class Linear-Transformation, which establishes a common protocol and default methods. The Matrix-Transformation class provides the canonical representation, in essence just wrapping a Common Loops object around a two-dimensional Common Lisp array kept in the matrix slot. Upper-Triangular-Transformation inherits the matrix slot from Matrix-Transformation, but its methods assume that the subdiagonal elements are zero to save roughly half the work in typical operations.

Vectors are represented in Cactus by one-dimensional Common Lisp arrays of numbers.

The algebra of linear transformations on a vector space requires four operations: scalar multiply, vector transform, compose, and sum. The first three are represented by a generic multiply function: mul. The sum of linear transformations is represented by the generic function: sum. Figure 4 compares the methods for transforming a vector, which are identical, except that the inner loop for Upper-Triangular-Transformation avoids unnecessary multiplies and adds of 0. (The iterations are expressed using the LOOP macro of Burke and Moon[4].)

In Cactus, we can solve an equation like $Ax = b$, not as a system of linear equations, but as a vector equation, $x \leftarrow A^{-1}b$ (for square and invertible A's). To do this, we represent A^{-1} by an instance of an appropriate inverse transformation class. Figure 5 shows the result for the data from figure 1.

```
(defclass Matrix-Transformation

        ;; super classes
        (Linear-Transformation)

        ;; slots
        (matrix))
```

```
(defclass Upper-Triangular-Transformation

        ;; super classes
        (Matrix-Transformation)

        ;; slots
        ())
```

Figure 3: Two simple transformation classes

```
(defmethod mul ((tr Matrix-Transformation)
                (v Vector))

  (loop with r = (make-array (range tr))
        with m = (matrix tr)
        for i from 0 below (range tr)
        do (setf (aref r i)
                 (loop for j from 0 below (domain tr)
                       sum (* (aref m i j) (aref v j))))
        finally (return r)))
```

```
(defmethod mul ((tr Upper-Triangular-Transformation)
                (v Vector))

  (loop with r = (make-array (range tr))
        with m = (matrix tr)
        for i from 0 below (range tr)
        do (setf (aref r i)
                 (loop for j from i below (domain tr)
                       sum (* (aref m i j) (aref v j))))
        finally (return r)))
```

Figure 4: mul methods for two simple classes.

```
(setf ta (make-instance 'Matrix-Transformation :matrix a))
(setf ta-1 (inverse ta))

(mul ta-1 b) → #(0.0 0.9999205 100.000015 10000.0)

x            → #(0.0 1.0000000 100.00000 10000.0)
```

Figure 5: Solving $Ax = b$ in Cactus.

```
(defclass Inverse-Transformation ;; an abstract class

        ;; super classes
        (Linear-Transformation)

        ;; slots
        (sire))
```

Figure 6: An abstract inverse class.

The key point is the existance of the generic inverse function, which returns an inverse object appropriate for the type of transformation being inverted.

All inverse transformation classes inherit from the abstract class Inverse-Transformation (see figure 6). Inverse-Transformation provides one slot, named sire, which points to the transformation whose inverse this is.

Suppose we start with an upper triangular transformation. Then constructing an inverse is easy (see figure 7). The inverse method for Upper-Triangular-Transformation simply creates an instance of Inverse-of-Upper-Triangular, filling the sire slot with the argument of inverse.

So an instance of Inverse-of-Upper-Triangular is essentially just a pointer

```
(defmethod inverse ((self Upper-Triangular-Transformation))

  (make-instance 'Inverse-of-Upper-Triangular :sire self))

---

(defclass Inverse-of-Upper-Triangular

        :: super classes
        (Inverse-Transformation)

        :: slots
        ())

---

(defmethod initialize ((self Inverse-of-Upper-Triangular)
                       (init-list list))

  (setf (sire self) (getf init-list :sire)))
```

Figure 7: An instantiable inverse class.

```
(defmethod mul ((tr Inverse-of-Upper-Triangular)
                (v Vector))

  (loop with r = (make-array (range tr))
        with u = (matrix (sire tr))
        with n = (domain tr)

        :: back substitution loop:
        for i from (- n 1) downto 0
        do (setf (aref r i)
                 (/ (- (aref v i)
                       (loop for j from (+ i 1) below n
                             sum (* (aref u i j) (aref r j))))
                    (aref u i i)))

        finally (return r)))
```

Figure 8: The mul method for the inverse of upper triangular is back substitution.

to the original upper triangular transformation. What makes it an inverse are its methods—for example, the mul method shown in figure 8, which uses back substitution rather than matrix multiply.

To compute an inverse of a Matrix-Transformation, we factor it into a product of transformations (like Upper-Triangular-Transformation) that can each be inverted accurately and efficiently. For gaussian elimination with partial pivoting the factoring is

$$A = (P_0 G_0 P_1 G_1 \dots P_n G_n U),$$

so that

$$A^{-1} \leftarrow (U^{-1} G_n^{-1} P_n^{-1} \dots G_0^{-1} P_0^{-1}).$$

We have seen how to represent U^{-1}, the representations of inverses of gauss and pivot transformations are also straightforward.

```
(defclass Inverse-of-Matrix-Transformation

        :: super classes
        (Inverse-Transformation)

        :: slots
        (u-inverse
         l-factors))

---

(defmethod mul ((tr Inverse-of-Matrix-Transformation)
                (v Vector))

  (loop with r = (make-array (range tr))
        for factor in (l-factors tr)
        do (mul! factor r)
        finally (return (mul! (u-inverse tr) r)))
  )
```

Figure 9: An inverse class for general matrix transformations.

The Inverse-of-Matrix-Transformation class is shown in figure 9. It provides two slots, one to hold the inverse of the upper triangular factor and another to hold a list of the inverses of the gauss and pivot factors. The mul method is also shown. It simply iterates over the factors, using the generic mul! function to destructively modify the result vector, r, without allocating temporary vectors at each iteration. (This strategy only works in situations where all the factors are square.)

The actual factoring is done by the initialize method for Inverse-of-Matrix-Transformation, as shown in figure 10. The construction is iterative. At the ith step a pivot transformation is chosen that interchanges rows of A_{i-1} (to

improve the numerical stability of the next step):

$$A_i' \leftarrow P_i A_{i-1}.$$

Then a gauss transformation is chosen to zero the sub-diagonal elements of the ith column of A_i':

$$A_i \leftarrow G_i A_i'.$$

Each P_i and G_i affect only the lower right $(n-i+1) \times (n-i+1)$ block of A_i. At the end, $U \leftarrow A_{n-1}$ is upper triangular. The chosen P_i's and G_i's are actually the inverses of the factors of A, so they can be collected directly in a list of the factors of A^{-1}.

The actual implementation, shown in figure 10, differs from the description in one significant way. We don't wish to allocate two new linear transformation objects at each step (to hold A_i' and A_i) so we use mul!, which overwrites its second argument, to destructively convert a copy of the original A to triangular form.

A the end of the iteration, u is a Matrix-Transformation whose sub-diagonal elements are zero. To get the right kind of inverse, we need u to be an Upper-Triangular-Transformation. Fortunately, Common Loops allows us to change the class of the existing instance u.

This implementation is almost as efficient as the carefully coded Fortran version in Linpack, which uses subtle knowledge of where zeros are to save space and avoid zero adds and multiplies at each step of the iteration. The Cactus version gets similar savings in space and time in a more modular way. Every class is responsible for providing its instances with information that allows them to be used efficiently and every method (for mul for example) is responsible for using that information appropriately. For example, mul methods for Gauss-Transformation take advantage of implicit block structure to save time. An implementor of a decomposition can assume that operations between pre-defined transformation classes will be efficient and can ignore internal details. For example, we can expect to get an efficient QR decomposition (which is be a better choice than LU when A is not square) by simply replacing Gauss-Transformation's with Householder-Transformation's and need not worry about how internal details differ.

6 Implications

Although I have only discussed a single, somewhat simplified example, I hope to have made a convincing case that linear algebra is a natural domain

```
(defmethod initialize ((self Inverse-of-Matrix-Transformation)
                       (init-list list))

  (setf (sire self) (getf init-list :sire))

  (loop with u = (copy self)
        with g
        with p

        for i from 0 below (- (range self) 1)
        do (setf p (choose-pivot-transformation u i))
           (mul! p u)
           (setf g (choose-gauss-transformation u i))
           (mul! g u)
        collect p into l-facs
        collect g into l-facs

        finally
          (change-class u (class-named 'Upper-Triangular-Transformation))
          (setf (u-inverse self) (inverse u))
          (setf (l-factors self) l-facs)))
```

Figure 10: Gaussian elimination with partial pivoting.

for object-oriented programming. It seems reasonable to expect similar value in other numerical problems, like optimization, differential equations, and statistical data analysis.

A full understanding of the implications of numerical computing for the design of object-oriented programming systems requires a more complete implementation than the current version of Cactus. However some points are already clear.

6.1 A large class lattice

A comprehensive linear algebra system would contain a surprisingly large number of classes—whose relationships may not be completely modeled by an inheritance lattice. To get a rough estimate of how many classes there might be, we can list several groups:

- Storage classes determine the Common Lisp data structures used to hold the state of the transformation, like Matrix, Vector, List-of-Vectors, and so on.
- Sparsity classes describe a patterns of zeros and/or other constants within a given storage class. For example, Upper-Triangular-Transformation asserts that all sub-diagonal elements of a matrix are exactly zero, which is important to know in *mul* or *inverse*.
- Assertion classes declare that their transformations have a certain property that is important in choosing the algorithm for one or more methods. Some assertions are easy to verify once, like Symmetric, but expensive to verify repeatedly. Others are difficult to verify directly but may be known from outside information or deduced from the way the *transform* is created, like Positive-Definite.
- Elementary transformations are created for the effect of composing with a particular instance, like Gauss, Householder, Givens[8] transformations that are used in various triangularization algorithms.
- Inverse-Transformation is an example of classes whose instance have a special relationship to some other instance, and who might need to change in some way if their sire changes.

- A general algebraic expression class represents a linear transformation as the sums and products of other transformations. Most of Linpack and Eispack are devoted to replacing linear transformations by equivalent algebraic expressions in transformations that are easier to deal with. For example, *inverse* replaces a *Matrix-Transformation* by a product of transformations that are individually easy to invert. It would be very useful for the system to be able to deduce properties of an expression from the properties of its components, eg. the sum of symmetric transformations is symmetric.

- Other expression classes involve operations like direct sum and quotient[9] that correspond to block structure in matrices.

- If we have an explicit representation of more abstract vector spaces than $\mathcal{R}^n$, then we have the possibility of linear transformation classes that are distinguished by their range and domain spaces.

Simply counting up all possible mixtures of classes like PositiveDefinite-Square-Symmetric-Tridiagonal-Matrix leads to hundreds, if not thousands of classes.

The potential complexity of the class lattice is intimidating and might lead one to prefer the simplicity of two-dimensional arrays. However, that apparent simplicity is misleading; a system like Linpack must deal with as many types of linear transformations as an object-oriented system. The only difference is that, in Fortran, the existance of the types and the methods for using them are implicit in the code, rather than explicitly represented in defclass's and defmethod's.

A key issue in further development is designing a minimal and sufficient class lattice. Even so, there are likely to be more possible classes then there are instances at any time. It might conceivably be useful to provide automatic generation of mixture classes, which raises the issue of how to use the logical, mathematical structure relating the classes. For example, asking the system to produce a class the inherits from both *Triangular* and Symmetric ought to produce a Diagonal-Transformation class.

6.2 Multi-methods

The key operations in linear algebra (like mul and sum) dispatch on 2 or 3 arguments. The ability to dispatch on multiple arguments is one of

the more outstanding differences between Common Loops and most other object-oriented languages.

Because of the large number of classes there may be as many as 10^4–10^6 distinct methods for a given generic function like mul. Good programming tools are vital for merely keeping track, for example, of all the mul methods that have an Triangular-Transformation as one of their arguments. In particular we want to take advantage of any possible shortcuts for filling in method tables. Using class inheritance to provide defaults and more generally coercing a transformation to another class are traditional techniques[1]. There are also possibilities for using sophisticated method combination rules to logically deduce the correct method.

6.3 Changing classes

The Change-class operation provided in Common Loops is a natural operation for linear algebra. It is the implicit paradigm underlying Linpack and Eispack: replace a given representation of a linear transformation by an alternative that's more suitable. Change-class preserves "eq-ness," which is a close analogy to Linpack, where the original matrix is usually overwritten by its decomposition. Eq-ness preservation is essential if we allow destructive modification of the state of linear transformations, which is unavoidable, at least for the largest problems. Changing a single matrix element may also cause a linear transformation to gain or lose properties like symmetry or positive-definiteness, which should be reflected in the class of the transformation.

References

[1] ABELSON, H., SUSSMAN, G., AND SUSSMAN, J. (1985)
Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass.

[2] BOBROW, D.G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M., AND ZDYBEL, F. (1985)
COMMONLOOPS: Merging Common Lisp and object-oriented programming, Intelligent Systems Laboratory Series ISL-85-8, Xerox PARC, 3333 Coyote Hill Road, Palo Alto, Ca. 94304.

[3] BOBROW, D.G., KAHN, K., KICZALES, G., MASINTER, L., STEFIK, M., and ZDYBEL, F. (1986)
CommonLoops: Merging Lisp and Object-Oriented Programming, Proceedings OOPSLA'86 (SIGPLAN Notices 21: 11 p. 17-29).

[4] BURKE, G. and MOON, D. (1981)
LOOP Iteration Macro, MIT LCS TM-169.

[5] DONGARRA, J.J., MOLER, C.B., BUNCH, J.R., and STEWART, G.W. (1979)
LINPACK Users' Guide. SIAM, Philiadelphia.

[6] GARBOW, B.S., BOYLE, J.M., DONGARRA, J.J., and MOLER, C.B. (1977)
Matrix Eigensystem Routines—EISPACK Guide Extension. Springer-Verlag, Berlin.

[7] GOLDBERG, A. and ROBSON, D. (1983b)
Smalltalk-80. The Language and Its Implementation. Addison-Wesley, Reading, Mass.

[8] GOLUB, G.H. and VAN LOAN, C.F. (1983)
Matrix Computations. The Johns Hopkins UNiversity Press, Baltimore.

[9] HALMOS, P.R. (1958)
Finite-dimensional Vector Spaces. Van Nostrand, Princeton, New Jersey.

[10] KEENE, S.E., and MOON, D.A. (1985)
Flavors: Object-oriented programming on Symbolics computers, presented at the Common Lisp Conference, Boston, Ma., December 1985; Symbolics Inc., 11 Cambridge Center, Cambridge, Ma. 02142.

[11] MOON D.A. (1986)
Object-Oriented Programmin with Flavors, Proceedings OOPSLA'86 (SIGPLAN Notices 21: 11 p. 1-8).

[12] ROYLANCE, G. (1984)
Some Scientific Subroutines in Lisp, MIT AI Memo 774.

[13] SMITH, B.T., BOYLE, J.M., DONGARRA, J.J., GARBOW, B.S., IKEBE, Y., KLEMA, V.C. and MOLER, C.B. (1976)
Matrix Eigensystem Routines—EISPACK Guide. 2nd Edition. Springer-Verlag, Berlin.

[14] STEELE, G.L. (1984)
Common Lisp. The Language. Digital Press.

[15] STEWART, G.W. (1973)
Introduction to Matrix Computations. Academic Press, New York.

END

DATE

FILMED

MARCH

1988

DTIC